Jon Bowen

CS 3210

Exercise 7

Ch 6 – Problem Set #1

*What are the arguments for and against representing Boolean values as single bits in memory?*

In favor of storing Boolean values as a single bit, this choice in theory saves memory. Only a single bit is need to represent the data. However, arguing against this choice, most machines cannot address a single byte, so a software implementation is needed to store Boolean values this way. If these values are stored in the smallest addressable memory unit, no special implementation is needed, but more memory is used than is in theory necessary.

Ch 6 – Problem Set #2

*How does a decimal value waste memory space?*

A decimal value wastes memory because it often uses more bytes than necessary to represent real numbers. With Binary Coded Decimals, each digit is represented exactly typically using 4 bits. Theee bits are not sufficient to store digits 0-9 so at least 4 bits must be used. But 4 bits can represent 16 digits, so there is essentially storage available that is never used. Floating point numbers use every bit in their allotted size to store a value, though often the exact value cannot be represented. In this way use less space than decimal values.

Ch 6 – Problem Set #7

*What significant justification is there for the -> operator in C and C++?*

Two significant justifications for the -> operator in C and C++ are readability and write-ability. This operator adds to readability in that the alternate syntax (e.g. (\*pointer).Member) requires arguably more effort on the reader of code to understand the operation. “Okay…dereference the pointer to get the variable and then access a member of the variable. What was the variable pointed to again?” It is much easier to think of student->ID as meaning “get the ID.” In the case of write-ability, writing “->” at one place where the semantic meaning is easily interpreted is, first, fewer keystrokes, and, second, less error prone since the alternative syntax will more frequently lead to missing or misplaces parentheses or forgetting to dereference the pointer.

Ch 6 – Problem Set #10

*Multidimensional arrays can be stored in row major order, as in C++, or in column major order, as in Fortran. Develop the access functions for both of these arrangements for three-dimensional arrays.*

For a 0-based 3D array of dimensions n x m x p (e.g. int[][][] arr = new int[n][m][p]) :

Row major:

addr(arr[i][j][k]) = addr(three[0][0][0]) + (k\*(n + m) + (i \* n + j) \* element\_size

Column major:

addr(arr[i][j][k]) = addr(three[0][0][0]) + (k\*(n + m) + (j \* m + i) \* element\_size

Ch 7 – Problem Set #4

*Would it be a good idea to eliminate all operator precedence rules and require parentheses to show the desired precedence in expressions? Why or why not?*

This would not be a good idea because it would significantly reduce the readability or code written in such a language. A reader would be required to parse any complex expression to be sure of the calculation being made, unwinding what may be complex nests of precedence with many adjacent parentheses. This mental effort could be spared by relying on common mathematical conventions on precedence for many operators that are well understood by programmers. It would also reduce write-ability since in general more keystrokes would be required to write expressions as well as increased attention in writing otherwise more straightforward expressions. A good programmer would also need to devote additional effort in constructing heavily parenthesized expression that enhance readability.

Ch 7 – Problem Set #5

*Should C’s assigning operations (for example, +=) be included in other languages (that do not already have them)? Why or why not?*

Yes they should be included. In my experience, such assignment operators are commonly used in relatively simple statements where the brevity of this syntax enhances both readability and write-ability. Mentally it is much simpler to look at the right hand side of the operator and comprehend what that right hand side represents. When the form sum = sum + count; is used, it certainly takes more keystrokes to write, but it also slightly increases the mental load to reduce the whole statement to simply “okay, we’re just adding count to sum. Of course such syntax can be misused. For example I consider it bad practice to have the left hand side of such operators appearing on the right hand side expression. However, with good training, the benefits outweigh the harm.

Ch 7 – Problem Set #9

a.

(((a \* b)1 – 1)2 + c)3

b.

(((a \* (b – 1)1)2 / c)3 mod d)4

c.

(((a – b)1 / c)2 & ((((d \* e)4 / a)5 – 3)6)3)7

d.

((-a)1 **or** ((c = d)2 **and** e)3)4

e.

(((a > b)1 **xor** c)3 **or** (d <= 17)2)4

f.

(-(a + b)1)2

Ch 7 – Problem Set #10

*Show the order of evaluation of the expressions of Problem 9, assuming that there are no precedence rules and all operators associate right to left.*

a.

(a \* (b – (1 + c)1)2)3

b.

(a \* ((b – 1)2 / (c **mod** d)1)3)4

c.

((a – b)5 / (c & ((d \* e)2 / (a – 3)1)3)4)6

d.

(-(a **or** (c = (d **and** e)1)2)3)4

e.

(a > (b **xor** (c **or** (d <= 17)1)2)3)4

f.

-(a + b)1)2

Ch 7 – Problem Set #13

a.

sum1 = 5 + fun(&i);

(value at &i) = 14;

fun(&i) returns (3 \* (value at &i) - 1);

fun(&i) returns (3 \* 14 - 1);

fun(&i) returns 41;

sum1 = 5 + 41;

sum1 = 46;

sum2 = fun(&j) + (j / 2);

(value at &j) = 14;

fun(&j) returns (3 \* (value at &i) - 1);

fun(&j) returns (3 \* 14 - 1);

fun(&j) returns 41;

sum2 = 41 + (14 / 2);

sum2 = 41 + 7;

sum2 = 48;

sum1 = 46

sum2 = 48

b.

**We assume dereferencing is still higher precedence than right to left evaluation since otherwise there would be pointer arithmetic with no effect.**

sum1 = (i/2) + fun(&i);

(value at &i) = 14; // <-- assume dereference highest precedence

fun(&i) returns (3 \* (value at &i) - 1);

fun(&i) returns (3 \* (14 – 1));

fun(&i) returns 39;

sum1 = (i/2) + 39;

sum1 = (14/2) + 39;

sum1 = 7 + 39;

sum1 = 46;

sum2 = fun(&j) + (10 / 2);

sum2 = fun(&j) + 5;

(value at &j) = 14; // <-- assume dereference highest precedence

fun(&j) returns (3 \* (value at &i) - 1);

fun(&j) returns (3 \* (14 - 1));

fun(&j) returns (3 \* 13);

fun(&j) returns 39;

sum2 = 39 + 5;

sum2 = 44;

sum1 = 46

sum2 = 44

Ch 7 – Programming #2

**C++**

#include <stdio.h>

int funLTR(int \*k){

(\*k) += 4;

return 3 \* (\*k) - 1;

}

int funRTL(int \*k){

(\*k) += 4;

return 3 \* ((\*k) - 1);

}

int main()

{

printf("Left to right order\n");

int i = 10, j = 10, sum1, sum2;

sum1 = (i / 2) + funLTR(&i);

sum2 = funLTR(&j) + (j / 2);

printf("sum1 = %d\n", sum1);

printf("sum2 = %d\n", sum2);

printf("Simulate right to left order\n");

i = 10, j = 10;

sum1 = funRTL(&i) + (i / 2);

sum2 = (j / 2) + funRTL(&j);

// sum1 = (i / 2) + (funRTL(&i));

// sum2 = funRTL(&j) + ((j / 2));

printf("sum1 = %d\n", sum1);

printf("sum2 = %d\n", sum2);

return 0;

}

Program output:

Left to right order

sum1 = 46

sum2 = 48

Simulate right to left order

sum1 = 46

sum2 = 44

**Java**

public class Test {

public static int funLTR(int[] k){

k[0] += 4;

return 3 \* k[0] - 1;

}

public static int funRTL(int[] k){

k[0] += 4;

return 3 \* (k[0] - 1);

}

public static void main(String[] args) {

System.out.println("Left to right order");

int[] i = new int[] { 10 };

int[] j = new int[] { 10 };

int sum1, sum2;

sum1 = (i[0] / 2) + funLTR(i);

sum2 = funLTR(j) + (j[0] / 2);

System.out.println("sum1 = " + sum1);

System.out.println("sum2 = " + sum2);

System.out.println("Simulate right to left order");

i = new int[] { 10 };

j = new int[] { 10 };

sum1 = funRTL(i) + (i[0] / 2);

sum2 = (j[0] / 2) + funRTL(j);

System.out.println("sum1 = " + sum1);

System.out.println("sum2 = " + sum2);

}

}

Program output:

Left to right order

sum1 = 46

sum2 = 48

Simulate right to left order

sum1 = 46

sum2 = 44

Ch 9 – Problem Set #5

a.

If the parameters are passed by value, the swap function as no effect since nothing is returned. Copies of the parameters are made, manipulated, and then released when they fall out of scope.

After each of the 3 calls the values of value and list are:

value = 2

list = { 1, 3, 5, 7, 9 }

b.

After 1st call to swap:

value = 1

list = { 2, 3, 5, 7, 9 }

After 2nd call to swap:

value = 1

list = { 3, 2, 5, 7, 9 }

After 3rd call to swap:

value = 2

list = { 3, 1, 5, 7, 9 }

c.

Pass by value-result is another implementation of inout-mode (pass by reference from part b is another). The difference between the two is that for pass by value-result, the actual parameters are copied to the formal parameters which then act like local variables during method execution. Then, just before execution returns to the calling program, the values of the formal parameters are copied back to the actual parameters. To a calling program, the observed behavior is the same as pass by reference, so we will observe no difference.

After 1st call to swap:

value = 1

list = { 2, 3, 5, 7, 9 }

After 2nd call to swap:

value = 1

list = { 3, 2, 5, 7, 9 }

After 3rd call to swap:

value = 2

list = { 3, 1, 5, 7, 9 }

Ch 9 – Programming #1